John von Neumann Institute for Computing

Potential Performance Improvement of Collective Operations in UPC

Rafik A. Salama, Ahmed Sameh

published in

Parallel Computing: Architectures, Algorithms and Applications ,
C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr, F. Peters (Eds.),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **38**, ISBN 978-3-9810843-4-4, pp. 413-422, 2007.
Reprinted in: Advances in Parallel Computing, Volume **15**,
ISSN 0927-5452, ISBN 978-1-58603-796-3 (IOS Press), 2008.

http://www.fz-juelich.de/nic-series/volume38

Potential Performance Improvement of Collective Operations in UPC

Rafik A. Salama and Ahmed Sameh

Department of Computer Science
American University in Cairo
E-mail: {raamir, sameh}@aucegypt.edu

Advances in high-performance architectures and networking have made it possible to build complex systems with several parallel and distributed interacting components. Unfortunately, the software needed to support such complex interactions has lagged behind. The parallel language's API should provide both algorithmic and run-time system support to optimize the performance of its operations. Some developers, however, choose to play clever and start from the language's primitive operations and write their own versions of the parallel operations. In this paper we have used a number of benchmarks to test performance improvement over current Unified Parallel C (UPC) collective implementations and prove that in some circumstances, it is wiser for developers to optimize starting from UPC's primitive operations. We also pin point specific optimizations at both the algorithmic and the runtime support levels that developers could use to uncover missed optimization opportunities.

1 Introduction

Collective communication operations in many parallel programming languages are executed by more than one thread/process in the same sequence taking the same input stream(s) to achieve common collective work[1]. The collective operations can either be composed by developers using the primitive operation's API that the language provides, or by parallel programming language writers who provide API for effective implementations of these collective operations. The extra effort of the language writers is meant to provide ease of use for developer to just call the collective operation rather than rewriting them using several primitive operations, and to supply highly optimized collective operations at two separate levels of optimization:

- *System runtime optimization*: The runtime library provides optimization opportunities at both hardware and system software levels. These optimizations may result in, for example, native use of the underlying network hardware and effective calls to Operating systems' services.
- *Algorithmic Optimization*: The algorithm is the core for optimizing collective operations. The collective operations can be highly optimized with the best proven algorithms.

UPC, or Unified Parallel C, is a parallel extension of ANSI C which follows the Partitioned Global Address Space (PGAS) distributed shared memory programming model that aims at leveraging the ease of programming of the shared memory paradigm, while enabling the exploitation of data locality. UPC is implemented by many universities (Berkley, Michigan, George Washington, Florida), companies (HP, IBM, Cray) and open source community (GNU GCC Compiler, ANL)[2, 3]. According to a latest research comparing the

performance of various UPC implementations, it has been established that the Berkley implementation is currently the best implementation of UPC.

Assuming that the Berkley UPC collective operations are highly optimized (at both runtime support and algorithmic levels), we used them as a reference for comparison with the less optimized collective operations provided by Michigan University[3]. Then starting from Michigan implementation of UPC primitive operations that provides two techniques as options to handle input streams, Push (Slave Threads pushing data to the master thread or the master thread pushes data to the slave threads) and the Pull (Master thread pulling data from the slave threads, or slave threads pulls data from the master thread), we build collective UPC operations by applying both algorithmic and runtime support. Most of these optimizations are borrowed from similar MPI collective operations. We have investigated in many implementations of the collective operations applied to the distributed and shared memory and then selected as a start the LAM implementation of MPI[5]. We have implemented two versions of each Michigan UPC collection operation, one based on Michigan Push technique and another based on Michigan Pull technique and compared them to the current Berkley UPC collective optimizations.

Finally, new non-LAM algorithmic optimizations were tried for the intensive collective operation AllReduce. We have also identified some bit falls in the UPC runtime support that couldn't implement specific performance optimization techniques for AllExchange.

The rest of the paper is organized as follows; the next section describes the test bed of the performance comparison benchmarks, the Third section presents the experimental results of the benchmarks showing the potential performance enhancement over Berkeley UPC collectives, the Fourth section describes algorithmic non-LAM optimization of the UPC collectives, the final section is a conclusion of our work.

2 Comparison Test Bed

2.1 Cluster Configuration

The performance comparison of benchmarks was done on a cluster developed by Quant-X which is a 63 GFLOPs (TPP: Theoretical Peak Performance) supercomputing facility with 14 nodes dual Intel Pentium IV Xeon 2.2 GHz, with 512 MB memory, Intel 860 chipset, 36 GB SCA hard disk (for a total of 15*36 GB), CD-ROM, Floppy, Ikle graphics cards, and M3F Myrinet 2000 Fiber/PCI 200 MHz interface cards[6].

2.2 Software Configuration

The Berkley UPC with the GASNET is installed over the 14 nodes of the cluster described above and the LAM MPI is also installed over the 14 nodes. The Berkley UPC GASNET is configured to use both the SMP (2 processors in each node) and LAM MPI for communication between the nodes. Also the Michigan UPC that confirms to UPC V1.1 is installed on the cluster[7].

2.3 Benchmarks

2.3.1 NPB framework Tailored for Collective

The NAS parallel benchmarks (NPB) are developed by the Numerical Aerodynamic simulation (NAS) program at NASA Ames Research Center for the performance evaluation of parallel supercomputers[8]. The NPB performance measurement framework were used and tailored the "NPB IS" benchmark (since it involves integer operations) to focus only on the collective operation and measure their timing. This collective focus implementation simply took the major workload classes (i.e S, A, B), the general function of data preparations, the data validation functions and the time measurement methods then started putting instead of the normal IS computation another function that only executes a collective operation with the various workloads and processor numbers given. For example, to measure the UPC AllReduce; the function simply works on the array already prepared by the NPB2.4 framework with a simple collective reduction operation.

2.3.2 Collective Optimization measurement

The collective optimization measurement is a comparison between the execution time taken by the **native** Berkeley UPC collective operation provided by the language that contains the runtime performance optimization and the **primitive** reference implementation of the Michigan UPC provided in both the Push and the Pull versions with added LAM-MPI optimizations.

3 Experimental Result

The experimental results have shown surprises for both the Push and Pull techniques. Although we will be exploring all results, but in general the choice of the PUSH or PULL according to the collective operation shows a notable performance potential. This is the case since UPC has the ability to recognize local-shared memory accesses, and perform them with the same low overhead associated with private accesses. The local-shared memory accesses can be divided into two categories: thread local-shared accesses, and SMP local-shared accesses, when a thread accesses data that is not local to the thread, but local to the SMP. The latter requires that implementation details are exploited using run-time systems, while the former can be exploited by compilers. Another PUSH/PULL optimization is the aggregation of remote accesses to amortize the overhead and associated latencies. This is done using UPC block transfer functions. Due to UPC thread-memory affinity characteristic, UPC compilers can recognize the need for remote data access at compile time, thus provide a good opportunity for pre-fetching. In all the results below, the y-axis of the graphs is calculated by the following equation:

$$OptimizationPercentage = (P/O - 1)\% \tag{3.1}$$

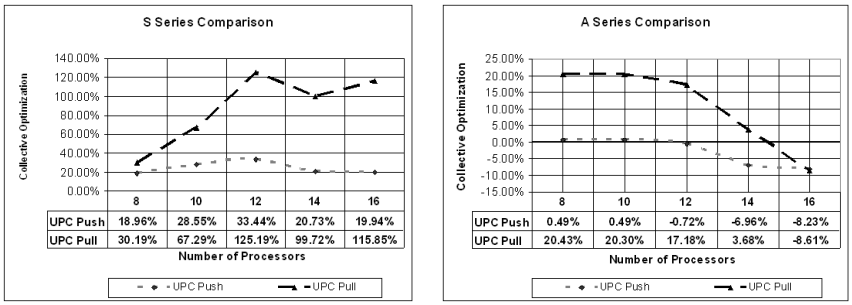
Where:

P	The primitive Michigan collective operations running Time
O	The native Berkely optimized collective operations running time.

Here the assumption is that collective operation should perform better than primitive operations which means higher than 1, so:

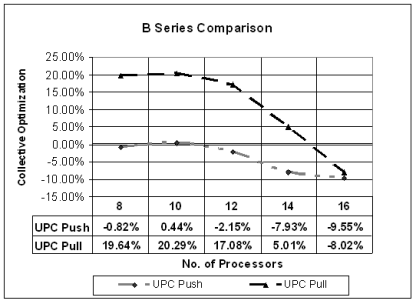
- The higher the value of the percentage above shows that the native Berkeley collective running time is lower than the Michigan primitive
- Zero means that native Berkeley collective is performing equal to the Michigan primitive operations
- Negative values indicate that the native Berkeley collective operations running time is higher than the Michigan primitive operations

Each point in the graphs below is an average of 600 actual result points for both the Michigan primitive collective as 300 point and the native Berkeley collective operations as 300 point.



(S)

(A)



(B)

Figure 1. AllGather Collective Optimization Comparison (Push & Pull vs. Native)

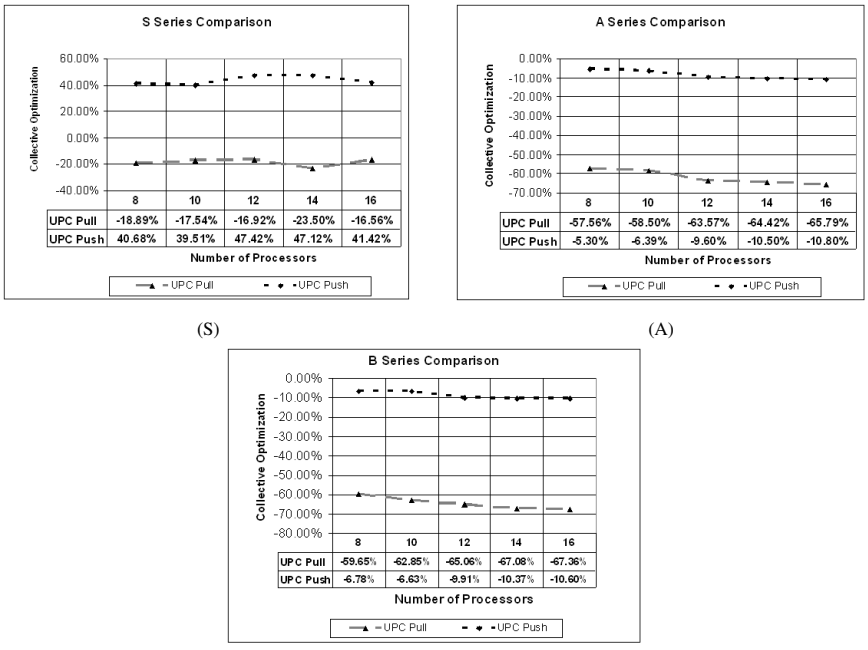
3.1 All Gather

ALL Gather was tested using the test bed described above showing as a general trend that the native Berkeley collective operations is better than the Michigan primitive collective operations with smaller data (S) as in Fig.1, but the performance kept getting worth with

larger data sizes (A,B) as in Fig.1. Also the comparison of the Push and Pull technique favoured the Push technique as expected, since the effort of sending the data to the parent process is distributed among all the slaves, rather than the Pull where the parent thread gets to do everything.

3.2 All Scatter

ALL Scatter testing results have shown an improvement by an average of 16 % for the pull technique in the small sizes S as shown in Fig.2. Over and above, the improvement has even become better with the larger sizes using the Pull technique as shown in Fig.2. This concludes that the Michigan primitive implementation using the Pull technique is better than the current Berkeley collective implementation. The Push technique on the other hand didn't show any enhancement over the current collective implementation and over the Pull technique which is more logical. A simple explanation is that the Pull technique divides the effort needed for data distribution among all the threads rather the Push technique which would have mandated for the parent thread to copy the data for the slave threads, rendering the parent thread a bottle neck in the collective operation.



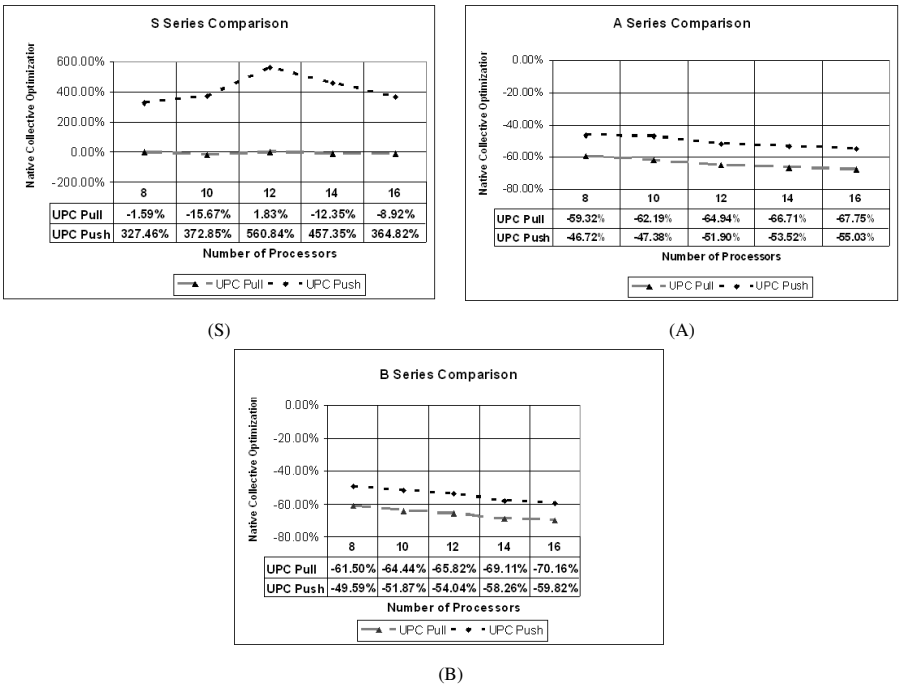
(S) (A)

(B)

Figure 2. AllScatter Collective Optimization Comparison (Push & Pull vs. Native)

3.3 All Broadcast

ALL Broadcast Michigan primitive native have generally shown better performance than the native Berkeley collective. The Push and Pull technique have shown that the Pull technique is much better than the Push technique in smaller sizes as in Fig.3, while the Push technique is almost the same as the Pull technique at higher sizes as in Fig.3.



(S) (A)

(B)

Figure 3. AllBroadcast Collective Optimization Comparison (Push & Pull vs. Native)

3.4 All Exchange

ALL Exchange have shown different behaviour at different sizes and different processors numbers. Initially the Push technique has shown better performance than the Pull technique at smaller data sizes (S) as shown Fig.3 and larger processor numbers (12 - 16). The Push technique on the other hand has shown better performance at larger data sizes (A, B) and larger processor numbers (12 - 16) as shown in Fig.4. This would conclude that generally the native Berkeley collective is performing better at small processor numbers, and there is a notable enhancement performance for the larger processors in the alternative use of the Push - Pull techniques according to data size.

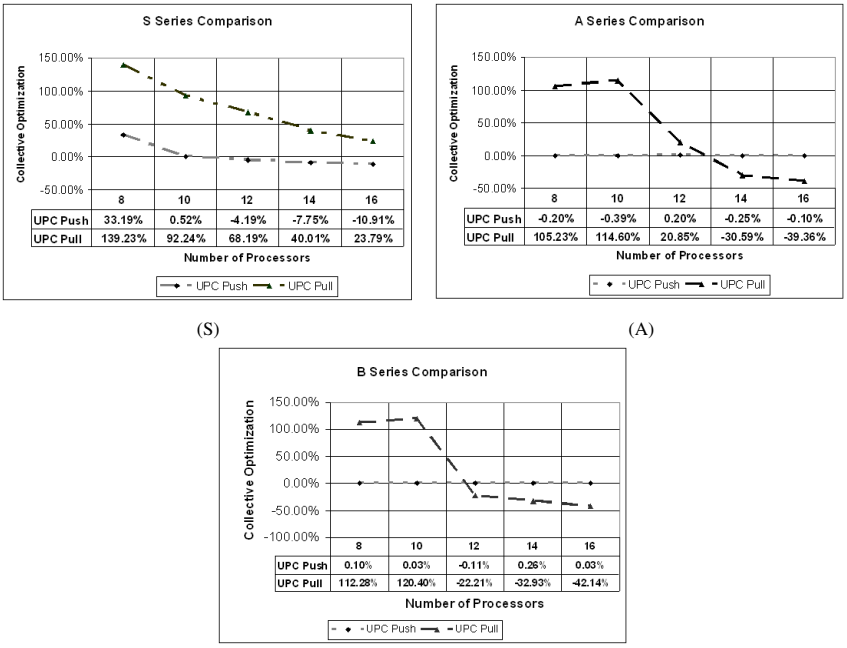


Figure 4. AllExchange Collective Optimization Comparison (Push & Pull vs. Native)

3.5 All Reduce

ALL Reduce Berkeley native collective operations have generally shown better performance than the Michigan primitive collective operations. The primitive collective operations have the worst performance in the small sizes (S), see Fig.5, while it gets better in the larger sizes (A, B) as shown in Fig.5.

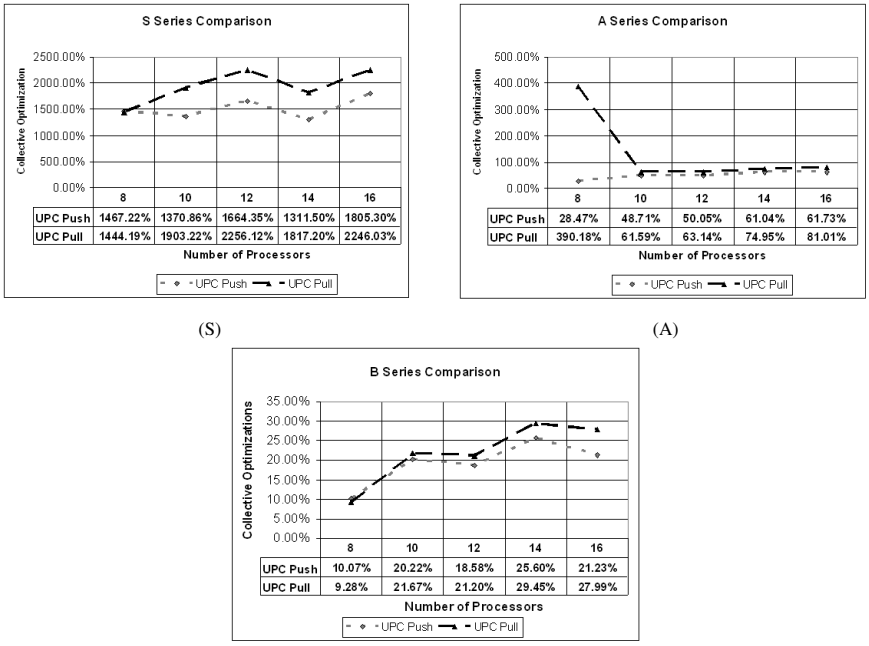


Figure 5. AllReduce Collective Optimization Comparison (Push & Pull vs. Native)

4 UPC Collective Operations Further Optimization

The native Berkeley UPC collective operations testing against the reference Michigan implementation have shown low performance in some operations while it has shown better performance in others. In our research we have explored the various optimizations done in the area of the collective operations enhancements and borrowed some of them to prove the room for enhancements. In this area, there have been many newly proposed algorithms as in[9–12] which offers a new set of algorithm for the MPI collectives as for example, the pipelining style which breaks the large messages into segments. These algorithms can be borrowed to the UPC collective implementations to enhance its current state. In the area of collective operation enhancements for the shared memory which is our focus, there have been many proposals as for the MPI Collective Algorithms over SMP[13] where the authors present an enhanced algorithms for the collective operations to provide a concurrent memory access feature, which would server a better performance as they have proved the efficiency of these algorithms to enhance the running time of the collective operations. The same algorithms could be borrowed for the UPC collective operations to enhance reflecting this in the compilation of the program or even reflecting it in the GASNET libraries in the SHM scheme. Also One of the modern techniques proposed in this area is the self tuned adaptive routines for the collective operations which delays the decision of the collective operation algorithm until the platform are known, this technique is called ”delayed finalization of MPI collective communication routines (DF)”[14], it

actually does allow for the MPI collective operations to detect the SMP architecture and automatically apply the needed algorithm for this architecture using the Automatic Empirical Optimization of Software (AEOS) technique[15].
In our comparison we borrowed as a start the optimizations done in LAM-MPI reference implementation since the MPI have shown better performance than the UPC in the collective operations[1]. Although MPI is a library while the UPC is a language (with its own compiler) but we have borrowed the optimizations of the MPI library to apply it for the UPC collective operations library.

In this section further optimization is done by applying further LAM MPI optimizations:

1. *Allgather & AllExchange*:

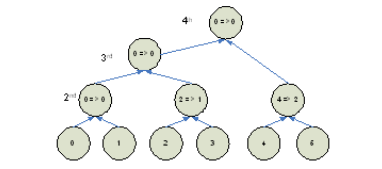
   Allgather & AllExchange are further investigated to borrow the same MPI Allgather & ALLtoALL optimization techniques respectively using the pairwise-exchange which have shown better performance[5], but we couldn't apply this optimization as this technique required the use of (asynchronous communication) which couldn't be simulated in the UPC due to lack in runtime support. As a matter of fact all the copying techniques in UPC are synchronous so it would have been so much helpful if the UPC supports asynchronous communication, which would offer apart from the collective operations a wealth of algorithms that uses the asynchronous communication.

2. *AllReduce*:

   ALLReduce on the other hand was highly optimized using a binary tree algorithm resulting in enhanced performance than the current collective operations.

   (a) *Algorithm*: the binary tree algorithm used is almost similar to the parallel binary tree algorithm[3], except for one fact, that the tree ranks is reconstructed again from the available nodes as shown in Fig.6

   (b) *Experimental Results*: the experimental results have shown enhancements over the current collective especially in the smaller sizes of data -S- as shown in Fig.7, and almost similar results with the higher sizes of data



While (Zero thread is not the last thread)
Compute current rank of the threads according to the step of the algorithm, if step zero, then nothing to be done.
Odd threads send their array to even threads.
Even threads receive odd array and compute the reduction with their own arrays.
Increment step by one
Continue

Figure 6. AllReduce Binary Tree Algorithm

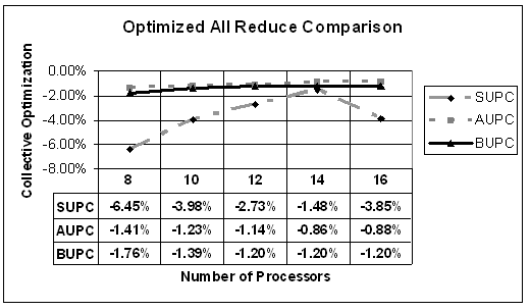


Figure 7. Optimized All Reduce Results

5 Conclusion

Group communications are commonly used in parallel and distributed environments. However, MPI collective communication operations have gone through several optimization enhancements. In this project, we borrow some of these optimization techniques into the UPC world. Experimental results show the borrowed techniques are solid and effective in improving UPC performance. The allreduce primitive collective was further optimized using a binary tree algorithm which showed better performance. So generally, there is a potential performance improvement and the current UPC Michigan implementation have shown an important need for the asynchronous memory communication where major MPI algorithms could be efficiently borrowed.

References

1. George Washington University and IDA Center for Computing Sciences, *UPC Consortium, UPC Collective Operations Specifications V1.0*, (2003).
2. Z. Zhang and S. Seidel, *Benchmark Measurements of Current UPC Platforms*, in: 19th IEEE International Parallel and Distributed Processing Symposium, IPDPS'05, (2005).
3. Michigan University UPC, *UPC Collective Reference Implementation V 1.0*, (2004), `http://www.upc.mtu.edu/collectives/coll.html`
4. George Washington University and IDA Center for Computing Sciences, *UPC Consortium, UPC Language Specifications V 1.2*, (2005).
5. J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel and J. J. Dongarra, *Performance analysis of MPI collective operations*, Cluster Computing, **10**, 127–143, (2007).
6. `http://www.cs.aucegypt.edu/ cluster`
7. `www.upc.mtu.edu/`
8. `www.nas.nasa.gov/Software/NPB/`
9. A. Faraj and X. Yuan, *Bandwidth efficient All-to-All broadcast on switched clusters*, in: 19th IEEE International Parallel and Distributed Processing Symposium, Boston, MA, (2005).
10. T. Kielmann, et. al., *MPI's collective communication operations for clustered wide area systems*, ACM SIGPLAN PPoPP, **1**, 131–140, (1999).
11. R. Rabenseifner, *A new optimized MPI reduce and allreduce algorithms*, (1997).
12. R. Thakur, R. Rabenseifner and W. Gropp., *Optimizing of collective communication operations in MPICH*, ANL/MCS-P1140-0304, Mathematics and Computer Science Division, Argonne National Laboratory, (2004).
13. S. Sistare, R. vandeVaart and E. Loh., *Optimization of MPI collectives on clusters of large scape SMPs*, in: Proc. High Performance Networking and Computing, SC'99, (1999).
14. S. S. Vadhiyar, G. E. Fagg and J. Dongarra, *Automatically tuned collective communications*, in: Proceedings of High Performance Networking and Computing, SC'00, (2000).
15. R. C. Whaley and J. Dongarra, *Automatically tuned linear algebra software*, in: High Performance Networking and Computing, SC'98, (1998).